

18 | 性能分析：找出程序的瓶颈

2020-06-16 罗剑锋

罗剑锋的C++实战笔记

[进入课程 >](#)



讲述：Chrono

时长 14:41 大小 13.46M



你好，我是 Chrono。

今天是“技能进阶”单元的最后一节课，我也要兑现刚开始在“概论”里的承诺，讲一讲在运行阶段我们能做什么。

运行阶段能做什么

在编码阶段，你会运用之前学习的各种范式和技巧，写出优雅、高效的代码，然后把它交给编译器。经过预处理和编译这两个阶段，源码转换成了二进制的可执行程序，就能够 CPU 上“跑”起来。



在运行阶段，C++ 静态程序变成了动态进程，是一个实时、复杂的状态机，由 CPU 全程掌控。但因为 CPU 的速度实在太快，程序的状态又实在太多，所以前几个阶段的思路、方法在这个时候都用不上。

所以，我认为，在运行阶段能做、应该做的事情主要有三件：**调试**（Debug）、**测试**（Test）和**性能分析**（Performance Profiling）。

调试你一定很熟悉了，常用的工具是 GDB，我在前面的“[🔗轻松话题](#)”里也讲过一点它的使用技巧。它的关键是让高速的 CPU 慢下来，把它降速到和人类大脑一样的程度，于是，我们就可以跟得上 CPU 的节奏，理清楚程序的动态流程。

测试的目标是检验程序的功能和性能，保证软件的质量，它与调试是相辅相成的关系。测试发现 Bug，调试去解决 Bug，再返回给测试验证。好的测试对于软件的成功至关重要，有很多现成的测试理论、应用、系统（你可以参考下，我就不多说了）。

一般来说，程序经过调试和测试这两个步骤，就可以上线运行了，进入第三个、也是最难的性能分析阶段。

什么是性能分析呢？

你可以把它跟 Code Review 对比一下。Code Review 是一种静态的程序分析方法，在编码阶段通过观察源码来优化程序、找出隐藏的 Bug。而性能分析是一种动态的程序分析方法，在运行阶段采集程序的各种信息，再整合、研究，找出软件运行的“瓶颈”，为进一步优化性能提供依据，指明方向。

从这个粗略的定义里，你可以看到，性能分析的关键就是“**测量**”，用数据说话。没有实际数据的支撑，优化根本无从谈起，即使做了，也只能是漫无目的的“不成熟优化”，即使成功了，也只是“瞎猫碰上死耗子”而已。

性能分析的范围非常广，可以从 CPU 利用率、内存占用率、网络吞吐量、系统延迟等许多维度来评估。

今天，我只讲多数时候最看重的 CPU 性能分析。因为 CPU 利用率通常是评价程序运行的好坏最直观、最容易获取的指标，优化它是提升系统性能最快速的手段。而其他的几个维度

也大多与 CPU 分析相关，可以达到“以点带面”的效果。

系统级工具

刚才也说了，性能分析的关键是测量，而测量就需要使用工具，那么，你该选什么、又该怎么用工具呢？

其实，Linux 系统自己就内置了很多用于性能分析的工具，比如 top、sar、vmstat、netstat，等等。但是，Linux 的性能分析工具太多、太杂，有点“乱花渐欲迷人眼”的感觉，想要学会并用在实际项目里，不狠下一番功夫是不行的。

所以，为了让你能够快速入门性能分析，我根据我这些年的经验，挑选了四个“高性价比”的工具：top、pstack、strace 和 perf。它们用起来很简单，而且实用性很强，可以观测到程序的很多外部参数和内部函数调用，由内而外、由表及里地分析程序性能。

第一个要说的是“**top**”，它通常是性能分析的“起点”。无论你开发的是什么样的应用程序，敲个 top 命令，就能够简单直观地看到 CPU、内存等几个最关键的性能指标。

top 展示出来的各项指标的含义都非常丰富，我来说几个操作要点吧，帮助你快速地抓住它的关键信息。

一个是按“M”，看内存占用（RES/MEM），另一个是按“P”，看 CPU 占用，这两个都会从大到小自动排序，方便你找出最耗费资源的进程。

另外，你也可以按组合键“xb”，然后用“<>”手动选择排序的列，这样查看起来更自由。

我曾经做过一个“魔改”Nginx 的实际项目，下面的这个截图展示的就是一次 top 查看的性能：

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
10380	root	20	0	3512m	2.6g	13m	S	105.9	4.2	0:55.15	nginx
10378	root	20	0	2065m	1.7g	2788	S	97.9	2.7	0:17.92	nginx
10466	root	20	0	1482m	1.0g	7932	S	78.2	1.7	0:19.92	nginx
10447	root	20	0	1482m	1.0g	7936	S	77.9	1.7	0:19.83	nginx
10434	root	20	0	1482m	1.0g	7936	S	77.2	1.7	0:19.83	nginx
10459	root	20	0	1482m	1.0g	7940	S	76.2	1.7	0:19.84	nginx
10384	root	20	0	1522m	1.1g	8528	S	70.2	1.8	0:28.31	nginx
10392	root	20	0	1523m	1.1g	8528	S	69.9	1.7	0:28.38	nginx
10402	root	20	0	1522m	1.1g	8512	S	69.9	1.8	0:28.45	nginx
10412	root	20	0	1522m	1.1g	8528	S	69.9	1.7	0:27.34	nginx
10425	root	20	0	1570m	1.1g	8528	S	69.9	1.7	0:27.46	nginx
10427	root	20	0	1522m	1.1g	8528	S	69.9	1.7	0:28.04	nginx
10381	root	20	0	1570m	1.1g	8528	S	69.6	1.7	0:28.78	nginx
10419	root	20	0	1570m	1.1g	8528	S	69.6	1.7	0:28.45	nginx
10377	root	20	0	1042m	813m	480	S	0.0	1.3	0:00.05	nginx
10564	root	20	0	99.9m	7660	492	S	0.0	0.0	0:00.00	nginx
10565	root	20	0	162m	63m	2200	S	0.0	0.1	0:00.36	nginx
10567	root	20	0	122m	27m	2164	S	0.0	0.0	0:00.13	nginx
10568	root	20	0	157m	58m	2228	S	0.0	0.1	0:00.83	nginx
10569	root	20	0	162m	63m	2204	S	0.0	0.1	0:00.73	nginx

从 top 的输出结果里，你可以看到进程运行的概况，知道 CPU、内存的使用率。如果你发现某个指标超出了预期，就说明可能存在问题，接下来，你就应该采取更具体的措施去进一步分析。

比如说，这里面的一个进程 CPU 使用率太高，我怀疑有问题，那我就要深入进程内部，看看到底是哪些操作消耗了 CPU。

这时，我们可以选用两个工具：**pstack** 和 **strace**。

pstack 可以打印出进程的调用栈信息，有点像是给正在运行的进程拍了个快照，你能看到某个时刻的进程里调用的函数和关系，对进程的运行有个初步的印象。

下面这张截图显示了一个进程的部分调用栈，可以看到，跑了好几个 ZMQ 的线程在收发数据：


```

Thread 5 (Thread 0x7f11b29fe700 (LWP 10386)):
#0 0x0000003f1c4e5d03 in epoll_wait () from /lib64/libc.so.6
#1 0x00007f11f4c6631d in zmq::epoll_t::loop() () from /usr/local/lib/libzmq.so.4
#2 0x00007f11f4c8651b in thread_routine () from /usr/local/lib/libzmq.so.4
#3 0x0000003f1c8077f1 in start_thread () from /lib64/libpthread.so.0
#4 0x0000003f1c4e570d in clone () from /lib64/libc.so.6
Thread 4 (Thread 0x7f11b1ffd700 (LWP 10387)):
#0 0x0000003f1c4e5d03 in epoll_wait () from /lib64/libc.so.6
#1 0x00007f11f4c6631d in zmq::epoll_t::loop() () from /usr/local/lib/libzmq.so.4
#2 0x00007f11f4c8651b in thread_routine () from /usr/local/lib/libzmq.so.4
#3 0x0000003f1c8077f1 in start_thread () from /lib64/libpthread.so.0
#4 0x0000003f1c4e570d in clone () from /lib64/libc.so.6
Thread 3 (Thread 0x7f11b15fc700 (LWP 10388)):
#0 0x0000003f1c4e5d03 in epoll_wait () from /lib64/libc.so.6
#1 0x00007f11f4c6631d in zmq::epoll_t::loop() () from /usr/local/lib/libzmq.so.4
#2 0x00007f11f4c8651b in thread_routine () from /usr/local/lib/libzmq.so.4
#3 0x0000003f1c8077f1 in start_thread () from /lib64/libpthread.so.0
#4 0x0000003f1c4e570d in clone () from /lib64/libc.so.6
Thread 2 (Thread 0x7f11b0bfb700 (LWP 10389)):
#0 0x0000003f1c4dc053 in poll () from /lib64/libc.so.6
#1 0x00007f11f4c79436 in zmq::signaler_t::wait(int) () from /usr/local/lib/libzmq.so.4
#2 0x00007f11f4c6a11e in zmq::mailbox_t::recv(zmq::command_t*, int) () from /usr/local/lib/libzmq.so.4
#3 0x00007f11f4c79dd4 in zmq::socket_base_t::process_commands(int, bool) () from /usr/local/lib/libzmq.so.4
#4 0x00007f11f4c7a01a in zmq::socket_base_t::recv(zmq::msg_t*, int) () from /usr/local/lib/libzmq.so.4
#5 0x00007f11f4c8e079 in s_recvmmsg () from /usr/local/lib/libzmq.so.4
#6 0x00007f11f4c8e5b2 in zmq_recv () from /usr/local/lib/libzmq.so.4

```

不过，pstack 显示的只是进程的一个“静态截面”，信息量还是有点少，而 strace 可以显示出进程的正在运行的系统调用，实时查看进程与系统内核交换了哪些信息：

```

epoll_wait(9, {{EPOLLIN|EPOLLOUT, {u32=3526323640, u64=139714517495224}}}, 512, 29) = 1
recvfrom(61, "$-1\r\n", 2048, 0, NULL, NULL) = 5
epoll_wait(9, {}, 512, 29) = 0
epoll_wait(9, {}, 512, 1) = 0
close(61) = 0
epoll_wait(9, {}, 512, 99) = 0
epoll_wait(9, {}, 512, 100) = 0
epoll_wait(9, {}, 512, 100) = 0
epoll_wait(9, {}, 512, 37) = 0
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 61
fcntl(61, FIONBIO, [1]) = 0
epoll_ctl(9, EPOLL_CTL_ADD, 61, {EPOLLIN|EPOLLOUT|EPOLLET|0x2000, {u32=3526323393, u64=139714517494977}}) = 0
connect(61, {sa_family=AF_INET, sin_port=htons(6379), sin_addr=inet_addr("127.0.0.1")}, 16) = -1 EINPROGRESS
epoll_wait(9, {{EPOLLOUT, {u32=3526323393, u64=139714517494977}}}, 512, 63) = 1
getsockopt(61, SOL_SOCKET, SO_ERROR, [-3300492014025441280], [4]) = 0
setsockopt(61, SOL_TCP, TCP_NODELAY, [1], 4) = 0

```

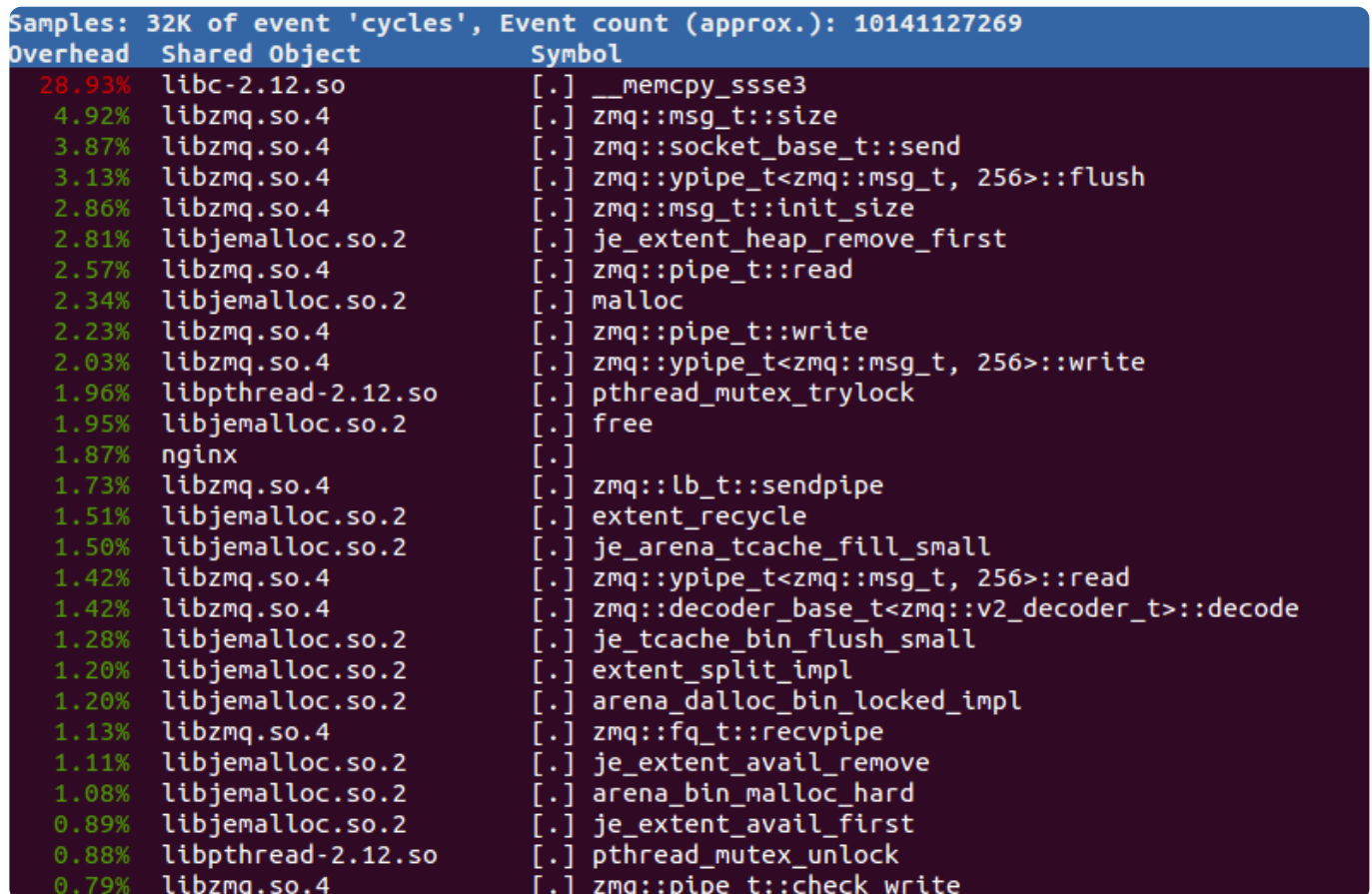
把 pstack 和 strace 结合起来，你大概就可以知道，进程在用户空间和内核空间都干了些什么。当进程的 CPU 利用率过高或者过低的时候，我们有很大概率能直接发现瓶颈所在。

不过，有的时候，你也可能会“一无所获”，毕竟这两个工具获得的信息只是“表象”，数据的“含金量”太低，做不出什么有效的决策，还是得靠“猜”。要拿到更有说服力的“数字”，就得 **perf** 出场了。

perf 可以说是 pstack 和 strace 的“高级版”，它按照固定的频率去“采样”，相当于连续执行多次的 pstack，然后再统计函数的调用次数，算出百分比。只要采样的频率足够大，把这些“瞬时截面”组合在一起，就可以得到进程运行时的可信数据，比较全面地描述出 CPU 使用情况。

我常用的 perf 命令是 “**perf top -K -p xxx**”，按 CPU 使用率排序，只看用户空间的调用，这样很容易就能找出最耗费 CPU 的函数。

比如，下面这张图显示的是大部分 CPU 时间都消耗在了 ZMQ 库上，其中，内存拷贝调用居然达到了近 30%，是不折不扣的“大户”。所以，只要能把这些拷贝操作减少一点，就能提升不少性能。



Samples: 32K of event 'cycles', Event count (approx.): 10141127269		
Overhead	Shared Object	Symbol
28.93%	libc-2.12.so	[.] __memcpy_ssse3
4.92%	libzmq.so.4	[.] zmq::msg_t::size
3.87%	libzmq.so.4	[.] zmq::socket_base_t::send
3.13%	libzmq.so.4	[.] zmq::ypipe_t<zmq::msg_t, 256>::flush
2.86%	libzmq.so.4	[.] zmq::msg_t::init_size
2.81%	libjemalloc.so.2	[.] je_extent_heap_remove_first
2.57%	libzmq.so.4	[.] zmq::pipe_t::read
2.34%	libjemalloc.so.2	[.] malloc
2.23%	libzmq.so.4	[.] zmq::pipe_t::write
2.03%	libzmq.so.4	[.] zmq::ypipe_t<zmq::msg_t, 256>::write
1.96%	libpthread-2.12.so	[.] pthread_mutex_trylock
1.95%	libjemalloc.so.2	[.] free
1.87%	nginx	[.]
1.73%	libzmq.so.4	[.] zmq::lb_t::sendpipe
1.51%	libjemalloc.so.2	[.] extent_recycle
1.50%	libjemalloc.so.2	[.] je_arena_tcache_fill_small
1.42%	libzmq.so.4	[.] zmq::ypipe_t<zmq::msg_t, 256>::read
1.42%	libzmq.so.4	[.] zmq::decoder_base_t<zmq::v2_decoder_t>::decode
1.28%	libjemalloc.so.2	[.] je_tcache_bin_flush_small
1.20%	libjemalloc.so.2	[.] extent_split_impl
1.20%	libjemalloc.so.2	[.] arena_dalloc_bin_locked_impl
1.13%	libzmq.so.4	[.] zmq::fq_t::recvpipe
1.11%	libjemalloc.so.2	[.] je_extent_avail_remove
1.08%	libjemalloc.so.2	[.] arena_bin_malloc_hard
0.89%	libjemalloc.so.2	[.] je_extent_avail_first
0.88%	libpthread-2.12.so	[.] pthread_mutex_unlock
0.79%	libzmq.so.4	[.] zmq::pipe_t::check_write

总之，使用 perf 通常可以快速定位系统的瓶颈，帮助你找准性能优化的方向。课下你也可以自己尝试多分析各种进程，比如 Redis、MySQL，等等，观察它们都在干什么。

源码级工具

top、pstack、strace 和 perf 属于“非侵入”式的分析工具，不需要修改源码，就可以在软件的外部观察、收集数据。它们虽然方便易用，但毕竟是“隔岸观火”，还是不能非常细致地分析软件，效果不是太理想。

所以，我们还需要有“侵入”式的分析工具，在源码里“埋点”，直接写特别的性能分析代码。这样针对性更强，能够有目的地对系统的某个模块做精细化分析，拿到更准确、更详细的数据。

其实，这种做法你并不陌生，比如计时器、计数器、关键节点打印日志，等等，只是通常并没有上升到性能分析的高度，手法比较“原始”。

在这里，我要推荐一个专业的源码级性能分析工具：**Google Performance Tools**，一般简称为 gperftools。它是一个 C++ 工具集，里面包含了几个专门的性能分析工具（还有一个高效的内存分配器 tcmalloc），分析效果直观、友好、易理解，被广泛地应用于很多系统，经过了充分的实际验证。

 复制代码

```
1 apt-get install google-perftools
2 apt-get install libgoogle-perftools-dev
```

gperftools 的性能分析工具有 CPUProfiler 和 HeapProfiler 两种，用来分析 CPU 和内存。不过，如果你听从我的建议，总是使用智能指针、标准容器，不使用 new/delete，就完全可以不用关心 HeapProfiler。

CPUProfiler 的原理和 perf 差不多，也是按频率采样，默认是每秒 100 次（100Hz），也就是每 10 毫秒采样一次程序的函数调用情况。

它的用法也比较简单，只需要在源码里添加三个函数：

ProfilerStart()，开始性能分析，把数据存入指定的文件里；

ProfilerRegisterThread()，允许对线程做性能分析；

ProfilerStop()，停止性能分析。

所以，你只要把想做性能分析的代码“夹”在这三个函数之间就行，运行起来后，gperftools 就会自动产生分析数据。

为了写起来方便，我用 shared_ptr 实现一个自动管理功能。这里利用了 void* 和空指针，可以在智能指针析构的时候执行任意代码（简单的 RAII 惯用法）：

 复制代码


```
1 auto make_cpu_profiler = // lambda表达式启动性能分析
2 [](const string& filename) // 传入性能分析的数据文件名
```

```

3 {
4     ProfilerStart(filename.c_str()); // 启动性能分析
5     ProfilerRegisterThread();        // 对线程做性能分析
6
7     return std::shared_ptr<void>(    // 返回智能指针
8         nullptr,                    // 空指针，只用来占位
9         [](void*){                  // 删除函数执行停止动作
10             ProfilerStop();          // 停止性能分析
11         }
12     );
13 };
14

```

下面我写一小段代码，测试正则表达式处理文本的性能：

 复制代码

```


1 auto cp = make_cpu_profiler("case1.perf"); // 启动性能分析
2 auto str = "neir:automata"s;
3
4 for(int i = 0; i < 1000; i++) {           // 循环一千次
5     auto reg = make_regex(R"^(^(\w+)\: (\w+)\$)"); // 正则表达式对象
6     auto what = make_match();
7
8     assert(regex_match(str, what, reg));    // 正则匹配
9 }
10

```

注意，我特意在 for 循环里定义了正则对象，现在就可以用 gperftools 来分析一下，这样做是不是成本很高。

编译运行后会得到一个“case1.perf”的文件，里面就是 gperftools 的分析数据，但它是二进制的，不能直接查看，如果想要获得可读的信息，还需要另外一个工具脚本 pprof。

但是，pprof 脚本并不含在 apt-get 的安装包里，所以，你还要从 [GitHub](#) 上下载源码，然后用“--text”选项，就可以输出文本形式的分析报告：

 复制代码

```

1 git clone git@github.com:gperftools/gperftools.git
2
3 pprof --text ./a.out case1.perf > case1.txt
4
5 Total: 72 samples

```



```
6  4  5.6%  5.6%  4  5.6% __gnu_cxx::__normal_iterator::base
7  4  5.6% 11.1%  4  5.6% _init
8  4  5.6% 16.7%  4  5.6% std::vector::begin
9  3  4.2% 20.8%  4  5.6% __gnu_cxx::operator-
10 3  4.2% 25.0%  5  6.9% std::__distance
11 2  2.8% 27.8%  2  2.8% __GI___strnlen
12 2  2.8% 30.6%  6  8.3% __GI___strxfrm_l
13 2  2.8% 33.3%  3  4.2% __dynamic_cast
14 2  2.8% 36.1%  2  2.8% __memset_sse2
15
```

pprof 的文本分析报告和 perf 的很像，也是列出了函数的采样次数和百分比，但因为是源码级的采样，会看到大量的内部函数细节，虽然很详细，但很难找出重点。

好在 pprof 也能输出图形化的分析报告，支持有向图和火焰图，需要你提前安装 Graphviz 和 FlameGraph：

📋 复制代码

```
1 apt-get install graphviz
2 git clone git@github.com:brendangregg/FlameGraph.git
```

然后，你就可以使用 “--svg” “--collapsed” 等选项，生成更直观易懂的图形报告了：

📋 复制代码

```
1 pprof --svg ./a.out case1.perf > case1.svg
2
3 pprof --collapsed ./a.out case1.perf > case1.cbt
4 flamegraph.pl case1.cbt > flame.svg
5 flamegraph.pl --invert --color aqua case1.cbt > icicle.svg
```

我就拿最方便的火焰图来“看图说话”吧。你也可以在 [GitHub](#) 上找到原图。

好了，今天主要讲了运行阶段里的性能分析，它能够回答为什么系统“不够好”（not good enough），而调试和测试回答的是为什么系统“不好”（not good）。

简单小结一下今天的内容：

1. 最简单的性能分析工具是 top，可以快速查看进程的 CPU、内存使用情况；
2. pstack 和 strace 能够显示进程在用户空间和内核空间的函数调用情况；
3. perf 以一定的频率采样分析进程，统计各个函数的 CPU 占用百分比；
4. gperftools 是“侵入”式的性能分析工具，能够生成文本或者图形化的分析报告，最直观的方式是火焰图。

性能分析与优化是一门艰深的课题，也是一个广泛的议题，CPU、内存、网络、文件系统、数据库等等，每一个方向都可以再引出无数的话题。

今天介绍的这些，是我挑选的对初学者最有用的内容，学习难度不高，容易上手，见效快。希望你能以此为契机，在今后的日子里多用、多实际操作，并且不断去探索、应用其他的分析工具，综合运用它们给程序“把脉”，才能让 C++ 在运行阶段跑得更好更快更稳，才能不辜负前面编码、预处理和编译阶段的苦心与努力。

课下作业

最后还是留两个思考题吧：

1. 你觉得在运行阶段还能够做哪些事情？
2. 你有性能分析的经验吗？听了今天的这节课之后，你觉得什么方式比较适合自己？

欢迎你在留言区写下你的思考和答案，如果觉得今天的内容对你有所帮助，也欢迎分享给你的朋友。我们下节课见。

课外小贴士

1. perf和gperftools的性能分析基于“采样”，所以数据只具有统计意义，每次的分析结果不可能完全相同，只要数据大体上一致，就没有问题。
2. GCC/Clang内置了Google开发的Sanitizer工具，编译时使用“-fsanitize=address”就可以检查可能存在的内存泄漏。
3. 火焰图由Brendan Gregg发明，把函数堆栈折叠为“可视化”的集合，从全局视角查看整个程序的调用栈执行情况。因为它像是一簇簇燃烧的火苗，所以被称为“火焰图”。
4. 基于动态追踪技术和火焰图，OpenResty公司开发出了全新的性能分析工具“OpenResty XRay”，看网站介绍，功能非常强大，感兴趣的话，可以去申请试用。

更多课程推荐

MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



涨价倒计时 🕒

今日秒杀 **¥79**，6月13日涨价至 **¥129**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 17 | 脚本语言：搭建高性能的混合系统

下一篇 轻松话题（一） | 4本值得一读再读的经典好书

精选留言 (5)

写留言



EncodedStar

2020-06-16

老师，火焰图可以在已经运行的程序中画出来吗？我看到命令都是./a.out，而且这个程序运行一个while true的话是不是图片特别大？

有没有像perf top -K -p xxx 这样的，直接观察在执行的进程

展开 ∨

作者回复: 可以向运行的进程发信号，然后在代码里收到信号后调用ProfilerStop，这样就可以随时生成火焰图。

或者用systemtap，不需要加gperftools代码，也可以生成火焰图，可以参考一下小贴士里的openresty xray。



1



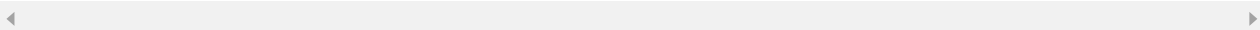
EncodedStar

2020-06-16

推荐大家可以看看极客时间倪鹏飞老师的linux性能分析系统实战老师这节总结的确实好，再次感谢老师～

展开 ▾

作者回复: 性能分析范围太大，一节课的内容只能结合C++讲最实用的几个技巧，大家后面可以深入研究。



1

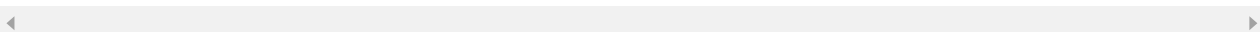


lckfa李钊

2020-06-16

Windows下的使用 wpr 和 wpa，通过pdb文件直达堆栈，也很方便

作者回复: Windows很久没用了，欢迎经验分享。



1



阿太

2020-06-16

valgrind-callgrind + kcache-grind = 电路图 也是利器

展开 ▾

作者回复: valgrind我用的比较少，对systemtap和火焰图更喜欢一些。



木瓜777

2020-06-16

`pprof --text ./a.out case1.perf > case1.txt`

执行pprof为什么需要 ./a.out 执行文件？ 如果执行文件要带参数才能运行，如何处理？

展开 ▾

作者回复: 它需要读取可执行文件里的符号才能产生分析文件，不需要运行时的参数。

